# Internship Report - Laboratoire Jean Kuntzmann

Galileo Grey

July 2023

## 1 Presentation of the Laboratory

The "Laboratoire Jean Kuntzmann" is a computing and applied mathematics laboratory supported by Grenoble Alpes University (UGA), Grenoble INP, as well as national research organisations CNRS and Inria.
The Lab is in the IMAG building on the Saint-Martin-d'Heres campus, which it shares with other labs such as Verimag and Grenoble Informatics Laboratory.
The laboratory hosts researchers, professors, PhD students, and especially in the summmer, interns. Most of the interns are graduate students. The lab convenes seminars, talks by local and visiting professors, and PhD defense hearings. The schedule of events is available on the lab's website. While at the lab, I attended a seminar about the compression of the mesh used for the simulation of geologically realistic models, as well as a PhD defense, presented by a student from the lab and examined by researchers from the CNRS and one from NYU, about the use of a complex filter to improve invariance by translation in machine learning algorithms for computer vision. I shared an office and socialised with other applied mathematics interns and PhD students, who were working on projects like applying machine learning technology to finance, creating statistics applications using R and Shiny, and studying statistical models.

## 2 Personal Assessment of the "Stage d'Excellence" Program

I was initially worried that as a first year undergraduate student there wouldn't be much I could contribute to research, especially in mathematics. While I don't think I was able to contribute to research in any meaningful capacity, I was able to see and experience the environment of an applied mathematics laboratory, work and speak with academics. More importantly, I was able to explore and understand more advanced mathematics research, interacting with more advanced and specific concepts than I would typically see in my university courses. I received guidance on how best to understand and implement the concepts I was learning. I learned both how to interact with research on my own and how to better explain and present my own work. I think the program

accomplishes its goal of giving students a taste of life in academia, and I enjoyed my experience thoroughly.

# 3 Theoretical and Experimental Results

This report was produced as part of internship with the goal of understanding and implementing concepts from chapters I and II of the book "Analyse Numérique et Équations Différentielles" by Jean-Pierre Demailly [1]. As such, most of the theoretical results below are taken directly or derived from the contents of those chapters.

The implementations of the concepts shown and the generation of the images contained in this document are available here (github.com/GreyGalileo/LJK-Internship).

## 3.1 Errors In Floating Point Arithmetic

Real numbers are often encoded on computers as floating point numbers, usually on either 32 or 64 bits. Floating point numbers have limited precision and can only encode a subset of rational numbers. When encoding real numbers, the floating point number used is often an approximation.

Floating point numbers are stored as a sign bit, an exponent, and a mantissa, where the exponent controls the order of magnitude of the number and the mantissa encodes the number's digits. When encoding floats in binary, since the first digit of the mantissa cannot be 0 it is always 1, and it can therefore be omitted, giving 1 extra bit of precision.

A binary floating point number is expressed as $x = (-1)^s \cdot m \cdot 2^p$, where $s$ is the sign bit, $m$ is the mantissa, and $p$ is the exponent. A floating point number of this form is necessarily constrained to the interval $]-2^{p_{max}+1}, 2^{p_{max}+1}[$ where $p_{max}$ is the maximum value the exponent can take on.

Under the IEEE standard, a 64 bit or "double precision" floating point number has an 11 bit exponent and a 52 bit mantissa with 1 bit for the number's sign, this allows for 53 significant binary figures, or approximately 16 significant decimal figures [2].

The approximation error refers to the difference between a real number $x$ and the value of the floating point number that encodes that real number. We denote this approximation error $\Delta x$. For any real $x \in ]-2^{p_{max}+1}, 2^{p_{max}+1}[$ and its associated floating point representation $\text{fp}(x)$, $\Delta x = |x - \text{fp}(x)|$

In general, assuming $x$ within the interval of representable floats, the error in floating point approximation is less than the value of the last bit of the mantissa. Since the value of this bit is determined by the order of magnitude of $x$ we can establish an upper bound on the relative approximation error:

$$\frac{\Delta x}{x} \leq \frac{2^{p-N}}{2^p} = 2^{-N}$$

where N is the size of the mantissa.

We call this upper bound $\varepsilon$ (sometimes called machine epsilon). In the case of IEEE standard 64 bit floats $\varepsilon = 2^{-52}$.

**Note.** This upper bound is specific to binary floats, in all other number bases, where a leading 1 cannot be assumed, the upper bound on approximation errors will be different ($b^{1-N}$ in base $b$ with $b \in \mathbb{N} \setminus \{2\}$).

**Note.** $\varepsilon$ can also be thought of as the difference between 1 and the smallest float $\geq 1$, and is a constant in the standard library of some programming languages [3].

When summing 2 floating point numbers, any digits after the last bit of the sum's mantissa are lost. The error produced when summing floats $x$ and $y$, which we denote $\Delta(x+y)$, is bounded above by $\varepsilon|x+y|$. Taking into account the errors in the floating-point representations of $x, y \in \mathbb{R}$, the upper bound becomes $\Delta x + \Delta y + \varepsilon|x+y|$

Similarly, when multiplying floats, all precision after the last bit of the product's mantissa is lost, the upper bound on the multiplication error $\Delta(xy)$ is $\varepsilon|xy|$, taking into account the errors on $x$ and $y$, this becomes $\Delta y|x| + \Delta x|y| + \varepsilon|xy|$ since the errors are multiplied by the full value of the other term.

Some processors have a dedicated fused multiply-add (fma) instruction which computes the expression $\text{fma}(x,a,b) = (xa) + b$ rounding only once, after the addition. The upper bound on the error is

$$\Delta(\text{fma}(x,a,b)) \leq \Delta a|x| + \Delta x|a| + \Delta b + \varepsilon(|xa+b|)$$

which is a lower upper bound in general than that of the computation without the fma instruction

$$\Delta((x \cdot a) + b) \leq \Delta a|x| + \Delta x|a| + \varepsilon|xa| + \Delta b + \varepsilon(|xa| + |b|)$$

This will be useful when evaluating polynomials in Section 3.2, which requires repeated addition and multiplication.

It should be noted that while the fused multiply-add circuit can improve the accuracy and performance of these operations, software emulation of the fma instruction is computationally costly. If the processor used to perform the algorithm does not have the fma instruction, performance may be significantly worse than that of the basic algorithm.

## 3.2 Numerical Evaluation of Polynomials

This section concerns two methods of evaluating a polynomial function $p$ of degree $n \in \mathbb{N}$ with coefficients $(a_i)_{0 \leq i \leq n}$ at any specified point $x$ on a certain finite interval $[a,b] \subset \mathbb{R}$. The polynomial can be expressed as $p(x) = \sum_{k=0}^{n} a_k x^k$.

The first "naive" method consists in recursively calculating $x^k$ by multiplying $x^{k-1}$ by $x$ and then multiplying that by its coefficient $a_k$ and adding it to a running sum, this means that at each iteration the algorithm performs 2 multiplications and 1 addition. Alternatively, adding $x^k a_k$ to the running sum

can be performed in 1 operation using the fma instruction described in Section 3.1.

The second method, called "Horner's Scheme" starts with $a_n$, the highest degree coefficient, instead of $a_0$, and at iteration $k$ multiplies the previous result by $x$ and adds $a_{n-k}$, thus each term $a_k$ is multiplied $k$ times by $x$. Again, the fma instruction can be used to multiply by $x$ and add $a_{n-k}$ with a single instruction.

Let $s_k$ be the intermediate result at iteration $k$, with $s_n$ being the final result, the above algorithms (naive, naive with fma, Horner's scheme, Horner with fma) can be expressed respectively as

$$\text{(N)} \begin{cases} s_0 = a_0 \\ x^k = x \cdot x^{k-1} \\ s_k = s_{k-1} + x^k \cdot a_k \end{cases}, \qquad \text{(N-fma)} \begin{cases} s_0 = a_0 \\ x^k = x \cdot x^{k-1} \\ s_k = \text{fma}(x^k, a_k, s_{k-1}) \end{cases},$$

$$\text{(H)} \begin{cases} s_0 = a_n \\ s_k = a_{n-k} + x \cdot s_{k-1} \end{cases}, \qquad \text{(H-fma)} \begin{cases} s_0 = a_n \\ s_k = \text{fma}(s_{k-1}, x, a_{n-k}) \end{cases}.$$

### 3.3 Example : Approximating the Error Function

The content of this Section is inspired by Chapter I Problem 4.1 in [1].

The error function is defined as

$$\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt.$$

We start by considering $\text{erf}(x)$ when $x \geq 0$. Let a function $g$ be defined as $g(x) = e^{x^2} \cdot \text{erf}(x)$, and, equivalently, $\text{erf}(x) = e^{-x^2} \cdot g(x)$. We know that $\text{erf}(0) = 0$, therefore $g(0) = 0$.
Computing the derivative of the error function gives:

$$\text{erf}'(x) = \frac{2}{\sqrt{\pi}} e^{-x^2} = e^{-x^2} \cdot g'(x) - 2x \cdot e^{-x^2} \cdot g(x),$$

where the second expression is the chain rule applied to $\text{erf}(x) = e^{-x^2} \cdot g(x)$.

$$\text{erf}'(x) = (e^{-x^2} \cdot g(x))^{(1)} = e^{-x^2} \cdot g'(x) - 2x \cdot e^{-x^2} \cdot g(x)$$

The function $g$ is therefore a solution of the differential equation

$$g'(x) = 2xg(x) + \frac{2}{\sqrt{\pi}}.$$

In particular, this means $g'(0) = \frac{2}{\sqrt{\pi}}$. Taking the derivative of that differential equation gives $g''(x) = 2xg'(x) + 2g(x)$. We can use the following recursive argument to find an expression for the $n$th derivative of $g$:

$$g^{(n)}(x) = 2xg^{(n-1)}(x) + 2(n-1)g^{(n-2)}(x) \implies g^{(n+1)}(x) = 2xg^{(n)}(x) + 2(n)g^{(n-1)}(x).$$

Since $g^{(2)}$ is of this form, for any $n > 1$, $g^{(n)}(x) = 2xg^{(n-1)}(x)+2(n-1)g^{(n-2)}(x)$ Evaluating the successive derivatives of $g$ at 0 gives $g$'s power series at 0:

$$g(x) = \sum_{n=0}^{+\infty} \frac{g^{(n)}(0)}{n!}x^n \text{ with } \begin{cases} g^{(0)}(0) = 0 \\ g^{(1)}(0) = \frac{2}{\sqrt{\pi}} \\ g^{(n)}(0) = 2(n-1)g^{(n-2)}(0), \ \forall n > 1 \end{cases} \tag{1}$$

Since at $x = 0$, $g^{(n)}$ is a multiple of $g^{(n-2)}$, $g^{(n)}(0)$ can be expressed as $g^{(n)}(0) = C_n g(0) = 0$ for all even values of $n$ and $g^{(n)}(0) = C_n g'(0) = C_n \frac{2}{\sqrt{\pi}}$ for all odd values of $n$, where $C_n \in \mathbb{N}$. At each iteration $C_n = 2(n-1)C_{n-2}$, $C_n$ is multiplied by 2 times the next even number. $C_n$ is therefore the product of all even numbers from 2 to $n-1$ and $2^{\frac{n-1}{2}}$ since $\frac{n-1}{2}$ is the number of iterations.

The product of all even numbers from 1 to $n-1$ can be expressed as $2^{\frac{n-1}{2}}(\frac{n-1}{2})!$. Therefore $C_n = 2^{\frac{n-1}{2}} \cdot 2^{\frac{n-1}{2}}(\frac{n-1}{2})! = 2^{n-1}(\frac{n-1}{2})!$ and

$$g^{(n)}(0) = \frac{2}{\sqrt{\pi}}2^{n-1}(\frac{n-1}{2})! = \frac{1}{\sqrt{\pi}}2^n(\frac{n-1}{2})!$$

for $n$ odd. Since $g^{(n)}(0) = 0$ for $n$ even:

$$\begin{aligned} g(x) &= \sum_{n=0}^{+\infty} \frac{g^{(n)}(0)}{n!}x^n = \sum_{n=0}^{+\infty} \frac{g^{(2n+1)}(0)}{(2n+1)!}x^{2n+1} \\ &= \sum_{n=0}^{+\infty} \frac{2}{\sqrt{\pi}} \frac{C_{2n+1}}{(2n+1)!}x^{2n+1} = \sum_{n=0}^{+\infty} \frac{1}{\sqrt{\pi}} \frac{2^{2n+1}n!}{(2n+1)!}x^{2n+1} \\ &= \sum_{n=0}^{+\infty} \frac{(2x)^{2n+1}n!}{\sqrt{\pi}(2n+1)!}. \end{aligned}$$

We can write $\operatorname{erf}(x)$ as $e^{-x^2}g(x)$ which is $e^{-x^2}\sum_{n=0}^{+\infty} \frac{(2x)^{2n+1}n!}{\sqrt{\pi}(2n+1)!} = \sum_{n=0}^{+\infty} \frac{(2x)^{2n+1}n!}{\sqrt{\pi}(2n+1)!}e^{-x^2}$. This gives an expression of the error function as the sum of a positive real sequence, we can write

$$\operatorname{erf}(x) = \sum_{n=0}^{+\infty} a_n(x) \text{ with } a_n(x) = \frac{(2x)^{2n+1}n!}{\sqrt{\pi}(2n+1)!}e^{-x^2},$$

$$\text{or } \begin{cases} a_0(x) = \frac{2x}{\sqrt{\pi}}e^{-x^2} \\ a_n(x) = \frac{(2x)^2 n}{2n(2n+1)}a_{n-1}(x) = \frac{2x^2}{2n+1}a_{n-1}(x). \end{cases}$$

The above recursion gives that $a_n(x)$ is decreasing as a function of $n$ when $2x^2 \leq 2n+1$ or equivalently, $\frac{2x^2}{2n+1} \leq 1$. Furthermore, $\frac{2x^2}{2n+1} \leq \frac{x^2}{n}$. Therefore, for some $N > x^2$,

$$\sum_{n=N+1}^{+\infty} a_n(x) < \sum_{k=1}^{+\infty} a_N(x) \left(\frac{x^2}{N}\right)^k = \frac{x^2}{N} \frac{a_N(x)}{1 - \frac{x^2}{N}} < a_N(x) \frac{x^2}{N - x^2}.$$

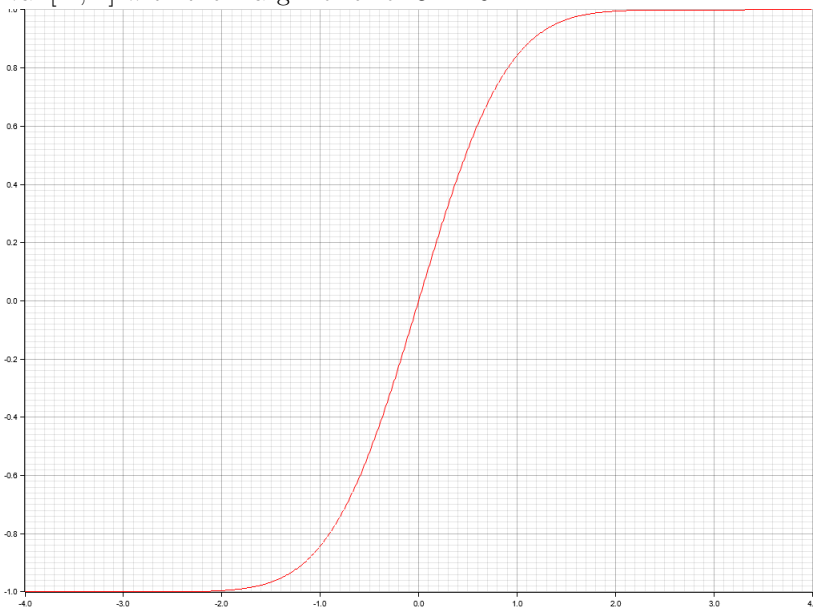
In terms of computation, the above result means that for any margin of error $\varepsilon > 0$ and a number of iterations $N \in \mathbb{N}$, if $a_N(x)\frac{x^2}{N-x^2} \leq \varepsilon$ then the result of the sum of all $a_n(x)$ with $n \leq N$ is within $\varepsilon$ of $\text{erf}(x)$, in other words

$$\sum_{n=0}^{N} a_n(x) < \text{erf}(x) < \sum_{n=0}^{N} a_n(x) + \varepsilon.$$

When computing the error function, we can end computation of the sum at a certain point based on the error allowed on the result.

While this algorithm only considers positive values of $x$, the error function is an odd function and so negative values can be computed by performing the algorithm on their absolute value then multiplying the result by $-1$.

Below is a graph of the error function computed using this algorithm on the interval [-4, 4] with the margin of error $\varepsilon = 10^{-14}$.




## 3.4 Polynomial approximations of functions using Lagrange Interpolation

Often it is useful to evaluate a polynomial approximation of a continuous function instead of the exact function itself. One way of creating such an approximation is to select a certain number of points of the original function and to build a Lagrange interpolating polynomial using those points.

The Lagrange method of interpolation, for a function $f$ and $n+1$ points labelled $(x_i)_{0 \leq i \leq n}$, consists in constructing a polynomial of degree $n$ called $l_i$

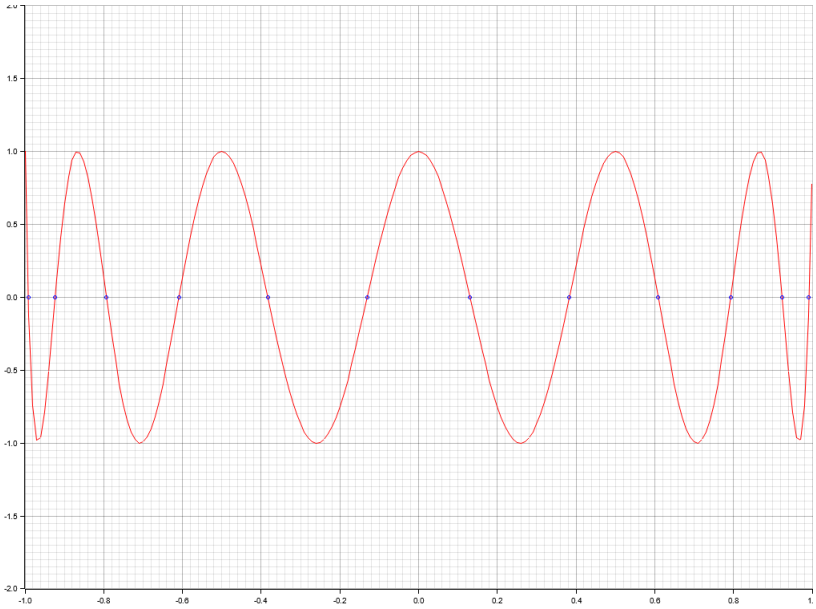
for each point $x_i$. This polynomial is defined as

$$l_i(x) = \Pi_{j \neq i} \frac{x - x_j}{x_i - x_j}.$$

As such for any $x_j$ where $j \neq i$, $l_i(x_j) = 0$ because it is multiplied somewhere by $x_j - x_j$ and $l_i(x_i) = 1$ because the top and bottom terms of each fraction in the product are the same. Multiplying $l_i$ by the constant $f(x_i)$ gives a polynomial of degree $n$ that takes on 0 at any $x_j$ except $x_i$, where it takes on the value of the desired function $f(x_i)$. The Lagrange interpolating polynomial is the sum of all the terms $f(x_i)l_i(x)$, which is a degree $n$ polynomial that takes on the value of $f$ at every point $x_i$.

Since the Lagrange interpolating polynomial interpolates points, not a function, the estimation error depends largely on the function and how much it oscillates between the points being interpolated. The formula for the upper bound on the error, assuming the function being estimated is differentiable at least $n+1$ times as given by [1] is, $||f - p_n|| \leq \frac{1}{(n+1)!} \cdot ||\pi_{n+1}|| \cdot ||f^{(n+1)}||$, where $p_n$ is the polynomial, $\pi_{n+1}(x) = \Pi_{j=0}^{n}(x - x_j)$ and $||f||$ is the uniform norm of a function on the interval in question, that is $||f|| = sup\{|f(s)| : s \in [a, b]\}$. The factor $||f^{(n+1)}||$ being included means that if the function oscillates very much between the interpolated points, the error will be greater.

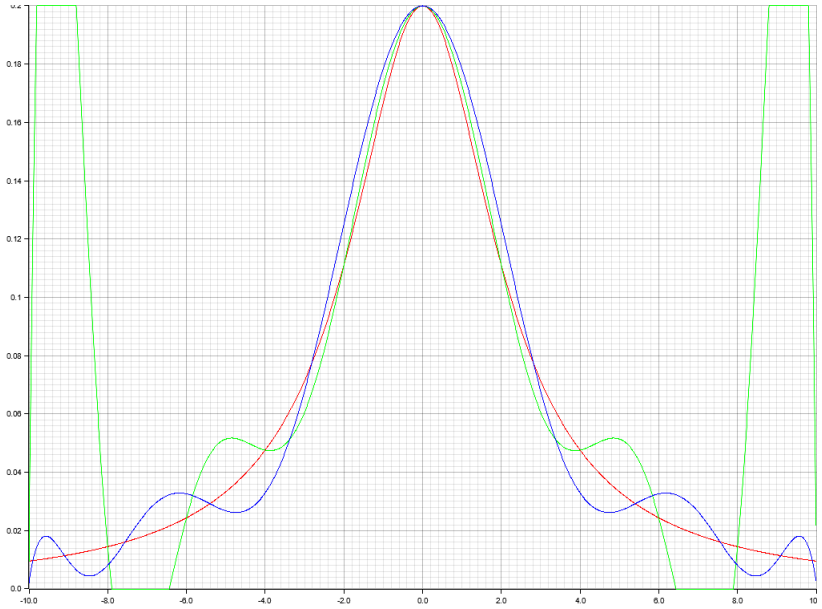
One way to reduce the upper bound on the error is to choose specific points instead of interpolating arbitrary points. One option is to interpolate using uniformly distributed points across the interval , another is to use the roots of the Tchebychev polynomial, defined as $t_n(x) = \cos(n \operatorname{Arccos}(x))$, where $n$ is the number of points. The Tchebychev polynomials only have roots (and are only defined) on the interval $[-1, 1]$, but the points can be trivially distributed across any interval $[a, b]$ using the linear map $x \mapsto \frac{a+b}{2} + \frac{b-a}{2}x$. Below is a graphic representation of a Tchebychev polynomial with $n = 12$ and its roots on the interval $[-1, 1]$.




According to [1],for $n$ points on a given interval $[a, b]$, $||\pi_{n+1}|| \leq (\frac{b-a}{e})^{n+1}$ if the points are uniformly distributed , and $||\pi_{n+1}|| \leq 2(\frac{b-a}{4})^{n+1}$ if the points are the roots of the Tchebychev polynomial.

The upper bound on the interpolation error is given as $||f-p_n|| \leq \frac{1}{(n+1)!}||\pi_{n+1}|| \cdot ||f^{(n+1)}||$. As such, for sufficiently large values of $n$, the upper bound on the interpolation error for equidistant points will be lower than for arbitrary points, and the upper bound for roots of Tchebychev polynomials will be lower than for equidistant points.

Below is an example of interpolating the function $f_\alpha(x) = \frac{1}{x^2+\alpha^2}$ (plotted in red) with $n = 11$ points on the interval $[-10, 10]$ using uniform distribution (green) and Tchebychev points (blue). This function is significant, according to [1] because, owing to Runge's phenomenon, it is an example of an analytic function for which the interploator using equidistant points does not converge when $n$ goes to infinity, even for high values of n. The figure clearly shows that the green curve oscillates near the interval boundaries, as would be expected.




Polynomial approximations such as the Lagrange interpolating polynomial are often useful, allowing for significant improvements in computation speed over other algorithms. However, as seen above, polynomial approximations can also lead to significant errors, even when selecting many points to interpolate. Understanding how interpolation and the functions being interpolated behave is vital in selecting methods of interpolation that ensure that the approximation remains useful, and that the polynomials in question converge to the desired function as the number of interpolated points grows.

# References

[1] Jean-Pierre Demailly. *Analyse numérique et équations différentielles - 4eme Ed.* EDP sciences, 2016. ISBN: 9782759819263.

[2] "IEEE Standard for Floating-Point Arithmetic". In: *IEEE Std 754-2019 (Revision of IEEE 754-2008)* (2019), pp. 1–84. DOI: 10.1109/IEEESTD.2019.8766229.

[3] The Rust Programming Language. *std - Rust*. URL: https://doc.rust-lang.org/beta/std/index.html. (accessed: 01.07.2023).